

This application is submitted in the name of inventors Zhiquan Chen and Judy Schwabe, assignors to Sun Microsystems, Inc., a Delaware Corporation.

SPECIFICATION

5

OPTIMIZATION OF N-BASE TYPED ARITHMETIC EXPRESSIONS

BACKGROUND OF THE INVENTION

10 Cross Reference to Related Applications

This application is related to the following:

U.S. Patent Application filed February 2, 1999, Susser and Schwabe, entitled
OBJECT-ORIENTED INSTRUCTION SET FOR RESOURCE-CONSTRAINED
DEVICES.

15 U.S. Patent Application filed April 15, 1997, Levy and Schwabe, entitled
VIRTUAL MACHINE WITH SECURELY DISTRIBUTED BYTECODE
VERIFICATION.

Field Of the Invention

20 The present invention relates to computer systems. More particularly, the present
invention relates to the optimization of n-base typed arithmetic expressions.

Background

Preparation of a computer program is illustrated in Fig. 1. The user writes programs in a high-level programming language 10. The programs written in the high-level program language 10 are compiled into a low-level machine language 12, which can be executed by the targeted machine. For example, programs written in the high-level Java™ programming language are compiled into low level bytecode instructions. The bytecode instructions are the machine language for a Java™ Virtual Machine. The Java™ Virtual Machine Specification is described in Lindholm et al., "The Java™ Virtual Machine Specification", 1999, Addison Wesley, Second Edition.

Typical high-level programming languages support arithmetic expressions. Arithmetic expressions are defined by an arithmetic operator that operates on one or more operands. Operators typically supported include addition, subtraction, multiplication, division, remainder, negate, shift, bitwise OR, bitwise AND and bitwise exclusive OR. Intermediate values are the results of one or more arithmetic operations.

High-level languages also typically support multiple or n-base integral types and arithmetic operations are overloaded. Overloading allows operators to accept operands having mixed types. For example, the Java™ programming language supports four base integral types: byte, short, int and long. These types support 8-, 16-, 32- and 64-bit values, respectively. Operators such as the "+" operator may accept operands of any of

these integral types. The three examples below illustrate overloading the “+” operator for operations on operands having mixed base types.

```
5  int a, b;  
    a+b;  
  
    short a, b;  
    a+b;  
  
10 byte a, b;  
    a+b;
```

This overloading is typically performed by widening values to a wider base type and then performing the arithmetic operation. For example, C and Java™ compilers typically widen values of type byte and short to type int. In the Java™ language, type int is always 32 bits. Thus, 16-bit values of type short and 8-bit values of type byte are widened to the 32-bit type int before performing the arithmetic operation. In the Java™ language, the following byte code is generated for each of the three examples listed above:

```
20  
  
    iload a  
    iload b  
    iadd
```

25 The iload instructions loads any of the 8, 16 or 32-bit variables and puts a 32-bit operand on the stack. The iadd instruction pops two 32-bit operands off the stack, adds them and puts the 32-bit result back on the stack.

Unlike Java™, some high-level languages define only the relationship between the integral types, and not the size of each type. For example, one C compiler vendor may define the bit sizes of types byte, short and int to be 8, 16 and 32 bits, respectively.

5 However, another C compiler vendor may define the sizes of the same types to be 16, 32 and 64 bits, respectively. Yet another compiler may define the bit sizes to be 16, 32 and 32 bits, respectively. In all cases, the relationship between the sizes of each type is maintained (number of values represented by type byte < number of values represented by type short, number of values represented by type short < number values represented
10 by type int), but the actual number of bits used to represent each type may differ. Like Java™, however, C performs arithmetic operations in the size of the int type defined by each particular compiler. This requires widening values having a smaller base type to type int.

15 This type widening approach reduces the number of machine instructions, thus reducing the complexity of the target machine. However, this type widening typically requires more computational stack space. For example, adding two 16-bit values of type short after they have been widened to the 32-bit type uses the same amount of stack space as adding two 32-bit values of type int, as illustrated in Figs. 2A and 2B.

20

Turning now to Fig. 2A, a flow diagram that illustrates stack usage when adding two 16-bit values of type short in the Java™ language is illustrated. At reference

numeral 20, the first 16-bit operand is loaded and pushed onto the operand stack. The operand stack at this point is illustrated by reference numeral 30. At reference numeral 22, the first 16-bit operand is expanded to 32 bits. At reference numeral 24, the second 16-bit operand is loaded and pushed onto the operand stack. At reference numeral 26, the second 16-bit operand is expanded to 32 bits. At this point, the operand stack occupies $4 \times 16 = 64$ bits. At reference numeral 28, the two 32-bit operands are added using a 32-bit add operator.

Turning now to Fig. 3A, a flow diagram that illustrates stack usage when adding two 32-bit values of type int is presented. At reference numeral 40, the first 32-bit operand is loaded and pushed onto the operand stack. The operand stack is illustrated by Fig. 3B. At reference numeral 42, the second 32-bit operand is loaded and pushed onto the operand stack. At reference numeral 44, the two 32-bit operands are added using a 32-bit add operator. Thus, in both the 16-bit add and the 32-bit add examples above, two 32-bit operands are pushed onto the stack before being popped off the stack and added using a 32-bit add operation.

During the course of program execution, the stack size may vary in size due to factors such as the level of nested procedure calls, the complexity of computed expressions and the number of locally declared variables. On resource-constrained devices such as smart cards, there is typically insufficient memory available to perform such computations where type widening takes place.

Resource-constrained devices are generally considered to be those that are relatively restricted in memory and/or computing power or speed, as compared to typical desktop computers and the like. By way of example, other resource-constrained devices include cellular telephones, boundary scan devices, field programmable devices, personal digital assistants (PDAs) and pagers and other miniature or small footprint devices.

Smart cards, also known as intelligent portable data-carrying cards, generally are made of plastic or metal and have an electronic chip that includes an embedded microprocessor or microcontroller to execute programs and memory to store programs and data. Such devices, which can be about the size of a credit card, have computer chips with 8-bit or 16-bit architectures. Additionally, these devices typically have limited memory capacity. For example, some smart cards have less than one kilo-byte (1K) of random access memory (RAM) as well as limited read only memory (ROM), and/or non-volatile memory such as electrically erasable programmable read only memory (EEPROM).

Furthermore, smart cards with 8-bit or 16-bit architectures typically have built-in 8-bit or 16-bit arithmetic operations, respectively. As such, smart cards can typically perform 8-bit or 16-bit operations more efficiently than 32-bit operations. Performing 32-bit operations on data that has been widened to 32-bits is especially inefficient. Thus, the limited architecture and memory of resource-constrained devices such as smart cards

make it impractical or impossible to execute programs where the values have been widened to a larger integral type.

The Java™ Virtual Machine instruction set defines an arithmetic instruction set to handle values of integral types byte, short and int. Variables of type byte and short are widened to the integral type int during compilation. By contrast, the Java Card™ (the smart card that supports the Java™ programming language) Virtual Machine defines a separate instruction set to handle variables of type byte and short, in addition to the instruction set to handle variables of integral type int. Most Java Card™ applications operate on data values of type short or byte.

There is an increasing trend in the computer industry to support high-level computer languages designed for execution on relatively memory-rich desktop computers, such that the same programs can be run on resource-constrained devices, thus achieving interoperability across vertical platforms. This interoperability across vertical platforms requires that programs written in the high-level programming language render the same result when run on resource-constrained devices as they would when ran on relatively memory-rich devices. For example, it is desirable to support execution of programs written in the Java™ programming language on a variety of platforms including smart card platforms, hand-held devices, consumer appliances, desktop computers and supercomputers.

Accordingly, there is a need to transform program representations such that semantically equivalent mathematical expressions can be performed using less computational stack space. Additionally, there is a need in the prior art to perform such transformations such that execution speed is increased.

SUMMARY OF THE INVENTION

A method for arithmetic expression optimization includes receiving a first instruction defined for a first processor having a first base, the first instruction including an operator and at least one operand, converting the first instruction to a second instruction optimized for a second processor having a second base when all operands do not carry potential overflow or when the operator is insensitive to overflow, the second base being smaller than the first base, and converting to a wider base a third instruction that is the source of the overflow when the at least one operand the potential for overflow and when the operator is sensitive to overflow. An apparatus for arithmetic expression optimization includes at least one memory having program instructions and at least one processor configured to use the program instructions to receive a first instruction defined for a first processor having a first base, convert the first instruction to a second instruction optimized for a second processor having a second base when every one of the at least one operand does not carry potential overflow or when the operator is insensitive to overflow, the second base being smaller than the first base, and convert to a wider base a third instruction that is the source of the overflow when the at least one operand the potential for overflow and when the operator is sensitive to overflow.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a block diagram that illustrates compiling a program written in a high-level language.

5

Fig. 2A is a flow diagram that illustrates stack usage for adding two 16-bit operands widened to 32-bits.

Fig. 2B is a block diagram that illustrates stack usage for adding two 16-bit operands

10 widened to 32-bits.

Fig. 3A is a flow diagram that illustrates stack usage for adding two 32-bit operands.

Fig. 3B is a block diagram that illustrates stack usage for adding two 32-bit operands.

15

Fig. 4A is a block diagram that illustrates converting arithmetic expressions for execution on a resource-constrained machine according to one embodiment of the present invention.

20 Fig. 4B is a block diagram that illustrates converting Java™ class files in accordance with one embodiment of the present invention.

Fig. 5A is a code sample that illustrates the addition of two values of type short on a desktop computer.

Fig. 5B is a code sample that illustrates the addition of two values of type short on a
5 resource-constrained computer.

Fig. 6A is a code sample that illustrates the addition of two values of type short and immediately casting the result on a desktop computer.

10 Fig. 6B is a code sample that illustrates immediately casting the result of an operation that potentially carries overflow on a resource-constrained computer.

Fig. 7A is a code sample that illustrates the addition of three values of type short and immediately casting the result on a desktop computer.

15

Fig. 7B is a code sample that illustrates performing an operation that is not affected by overflow on operands by an operation that the potential for overflow on a resource-constrained computer.

20 Fig. 8A is a code sample that illustrates the addition of two values of type short and dividing the result by a value of type short on a desktop computer.

Fig. 8B is a code sample that illustrates performing an operation that is affected by overflow on operands created by an operation that the potential for overflow on a resource-constrained computer.

- 5 Fig. 9 is a flow diagram that illustrates a method for n-base typed arithmetic expression optimization in accordance with one embodiment of the present invention.

Fig. 10 is a detailed flow diagram that illustrates a method for n-base typed arithmetic expression optimization in accordance with one embodiment of the present invention.

10

Fig 11 is a flow diagram that illustrates converting an instruction in accordance with one embodiment of the present invention.

- Fig. 12A is a flow diagram that illustrates a method for converting a target instruction in accordance with one embodiment of the present invention.
- 15

Fig. 12B is a flow diagram that illustrates a method for converting an initial value instruction in accordance with one embodiment of the present invention.

- 20 Fig. 13 is a flow diagram that illustrates a method for converting a type conversion instruction in accordance with one embodiment of the present invention.

Fig. 14 is a flow diagram that illustrates a method for converting a stack manipulation instruction in accordance with one embodiment of the present invention.

Fig. 15 is a flow diagram that illustrates a method for converting an arithmetic expression
5 in accordance with one embodiment of the present invention

Fig. 16 is a flow diagram that illustrates a method for determining an optimized instruction type in accordance with one embodiment of the present invention.

10 Fig. 17 is a flow diagram that illustrates a method for determining a result type and result overflow in accordance with one embodiment of the present invention.

Fig. 18 is a flow diagram that illustrates a method for recording a rollback point in accordance with one embodiment of the present invention.

15

Fig. 19 is a flow diagram that illustrates a method for rolling back the conversion process in accordance with one embodiment of the present invention.

Fig. 20 is a flow diagram that illustrates propagating the results of an instruction
20 optimization in accordance with one embodiment of the present invention.

Fig. 21 is a flow diagram that illustrates merging conversion information from different control paths in accordance with one embodiment of the present invention.

Fig. 22A is a block diagram that illustrates instruction conversion in accordance with one
5 embodiment of the present invention.

Fig. 22B is a block diagram that illustrates instruction conversion in accordance with one embodiment of the present invention.

DETAILED DESCRIPTION OF THE SPECIFIC EMBODIMENTS

Those of ordinary skill in the art will realize that the following description of the present invention is illustrative only. Other embodiments of the invention will readily suggest themselves to such skilled persons having the benefit of this disclosure.

This invention relates to computer systems. More particularly, the present invention relates to the optimization of n-base typed arithmetic expressions. The invention further relates to machine readable media on which are stored (1) the layout parameters of the present invention and/or (2) program instructions for using the present invention in performing operations on a computer. Such media includes by way of example magnetic tape, magnetic disks, optically readable media such as CD ROMs and semiconductor memory such as PCMCIA cards. The medium may also take the form of a portable item such as a small disk, diskette or cassette. The medium may also take the form of a larger or immobile item such as a hard disk drive or a computer RAM.

Resource-constrained devices are generally considered to be those that are relatively restricted in memory and/or computing power or speed, as compared to typical desktop computers and the like. Although the particular implementation discussed below is described in reference to a smart card, the invention can be used with other resource-constrained devices including, but not limited to, cellular telephones, boundary scan devices, field programmable devices, personal digital assistants (PDAs) and pagers, as

well as other miniature or small footprint devices. The invention can also be used on non-resource constrained devices.

For the purpose of this disclosure, the term "processor" may be used to refer to a
5 physical computer or a virtual machine.

Turning now to Fig. 4A, a block diagram that illustrates converting arithmetic expressions for execution on a resource-constrained machine according to one embodiment of the present invention is presented. A compiler takes arithmetic
10 expressions 60 written in a high-level language 62 and widens the operands to a larger integral type, creating larger base typed instructions 64 for execution on a typical desktop machine 66. The larger base typed instructions 64 are optimized to semantically equivalent smaller base typed instructions 68 for execution on a resource-constrained machine 70. For example, a short-type addition instruction is used to operate on short-
15 typed operands; and the result is type short.

According to another embodiment of the present invention, the optimization to semantically equivalent smaller base typed instructions is part of a just-in-time code generator. Just before a set of instructions is executed for the first time, the unoptimized
20 instructions are optimized to semantically equivalent smaller base typed instructions for execution on a resource-constrained machine. Subsequent execution of the same set of instructions use the set of optimized instructions.

According to another embodiment of the present invention, when a larger type instruction 64 is required to preserve the semantics of an arithmetic instruction, and larger type instructions are not supported by the target processor, the arithmetic expression is
5 rejected as not supported.

Turning now to Fig. 4B, a block diagram that illustrates converting instructions in accordance with one embodiment of the present invention is presented. Java™ class files 72 containing instructions with 32-bit operands are received by a Java Card™ class file
10 converter 74. The converter 74 generates instructions 76 optimized for execution on a resource-constrained device. The optimizations include, by way of example, providing less stack usage, smaller program size and faster execution.

Target machines may support n-typed arithmetic operators. While the Java™
15 Virtual Machine supports type int operators, the Java Card™ Virtual Machine supports type short operators and optionally supports type int operators. Other devices may support only byte-typed arithmetic operations, or all of byte-, short- and int-typed operations. Typically, relatively less time is required to perform 16-bit arithmetic on an 8-bit or 16-bit processor and relatively more time is required to perform 32-bit arithmetic
20 on the same processor.

Since the actual values used in an arithmetic operation are not known at optimization time, the optimization must assume the worst case value for each operand. The worst case value for an operand is determined based upon the input operand type. A small-type operation can have results that require large-type representation or overflow into a larger type. Thus, according to the present invention, arithmetic operators are categorized into operators affected by overflow and operators with the potential to create overflow. For the purposes of this disclosure, overflow includes the underflow of negative values. The result of a small-type operation is said to carry potential overflow if the operator used to create the result belongs to the group of operators with the potential to create overflow into a large-type representation. Intermediate values are allowed to carry potential overflow as long as the intermediate value is not used as an operand for an operator belonging to the group of operators affected by overflow.

The operators with a potential to create overflow include addition, subtraction, multiplication, division, negate and left shift. The Java™ bytecodes for these operators are shown in Table 1.

Java™ Bytecode
iadd
isub
imul
idiv
ineg
ishl

Table 1 – Operations with Potential Overflow

The operators affected by overflow are shown in Table 2. The arithmetic operators affected by overflow include division, remainder, negate, right-shift and unsigned right-shift. Non-arithmetic operators affected by overflow include array operations, switch operations and compare operations.

Java™ Bytecode	Operation Type	Which Operand(s) Affected by Overflow
idiv	Arithmetic	both input operands
irem	Arithmetic	both input operands
ineg	Arithmetic	only has one operand
ishr	Arithmetic	operand being shifted only
iushr	Arithmetic	operand being shifted only
if<*>	Compare	only has one operand
if_icmp<*>	Compare	both operands to the compare
tableswitch	Switch	switch value
lookupswitch	Switch	switch value
*newarray	array	number of elements
*aload	array	array index
*astore	array	array index

Table 2 – Operations Affected by Overflow

When optimizing Table 1 operations to a smaller type, the result may overflow into the larger type. The result of an expression with one of the operators in Table 2 may lose precision if one of the operands in the expression is an intermediate value and contains potential overflow data. To enable optimization and preserve the semantics of the high-level source code, the potential overflow must be corrected using an explicit

source level cast to the type of the result if the result is input to one of the operations in Table 2.

If input operand(s) to any of the operations in Table 2 are the result of an operation
5 in Table 1 and an explicit high level source code cast is not present, optimization cannot
occur. Such an erroneous optimization would not guarantee a semantically equivalent
result. In other words, the optimized code generated for execution on a resource-
constrained device could render a result different than the non-optimized code generated
for a desktop computer. For example, overflow data could be present in the Java™ 32-
10 bit representation of the operand(s), but not in the Java Card™ 16-bit representation.

The result of operations with the operators listed in Table 1 may cause overflow if
an operator with a smaller type is applied. Examples of these problems associated with
optimizing instructions targeted to a desktop computer platform to instructions targeted to
15 a resource-constrained computer platform are provided in Figs. 5A-8B. The examples
assume the desktop computer is based on a 32-bit architecture and is relatively memory
rich. The resource-constrained computer is assumed to be based on a 16-bit architecture
with relatively little memory. Those of ordinary skill in the art will recognize the
invention applies to computing platforms having various architectures.

Figures 5A-8B also use signed values. Those of ordinary skill in the art will also recognize that overflow may occur regardless of whether the values are signed or unsigned.

5 Turning now to Fig. 5A, a code sample that illustrates the addition of two values of type short on a desktop computer is illustrated. The value "a" contains the maximum value that can be represented by a 16-bit signed short type. As described above, even though the values are 16-bit short values, int-type addition is used. Thus, overflow from the 16-bit range to the 32-bit range is present in the result value and the effect of the
10 overflow is to create a larger positive 32-bit number.

Turning now to Fig. 5B, a code sample that illustrates adding the same values as in Fig. 5A on a resource-constrained computer is presented. Since execution is being performed on a resource-constrained computer and both values are 16-bit short types, the
15 instructions are optimized to use short-typed addition, thus using less stack space. However, because 16-bit addition is used instead of 32-bit addition, the addition creates overflow in the sign bit. Whereas the desktop computer computed a value of 32,768, the result computed in the resource-constrained computer example is -32,768, a negative number. This result is unacceptable because it is different from the desktop computer
20 result, preventing interoperability across multiple computer platforms.

Turning now to Fig. 6A, a code sample that illustrates the addition of two values of type short and immediately casting the result is presented. This example is the same as that in Fig. 5A, except that the result of the addition is cast to type short. Casting the type to short truncates the most significant sixteen bits to a short value and sign extends to a 32-bit value. The result of an operation that potentially carries overflow (the add operation) is cast to type short, thereby eliminating any potential overflow problem. Fig. 6B illustrates adding the same values as in Fig. 6A represented as 16-bit values on a resource-constrained computer. The result values for both the desktop computer and the resource-constrained computer are the same.

10

Turning now to Fig. 7A, a code sample that illustrates the addition of three values of type short on a desktop computer is presented. In the example, int-type addition is used to add 16-bit short values "a" and "b" and add the result to "c". The final result is cast to a short type.

15

Turning now to Fig. 7B, a code sample that illustrates performing an operation that is not affected by overflow on operands created by an operation that potentially carries overflow on a resource-constrained computer is presented. Since all values in this example are 16-bit short types, short-typed addition is used for all intermediate additions. As indicated in Table 1, the addition operator potentially creates overflow, but is not affected by overflow. Thus, adding "a" and "b" creates a value that potentially carries overflow. This value is added to "c", creating another value that potentially carries

20

overflow. Although the second add operation contains one operand that potentially carries overflow (the $a+b$ result), the add operation is not affected by operands carrying overflow. The final result is cast to type short, removing the potential overflow from the addition operation. Thus, the result values for both the desktop computer and the
5 resource-constrained computer are the same.

Turning now to Fig. 8A, a code sample that illustrates the addition of two values of type short and dividing the result by a value of type short on a desktop computer is presented. Since execution is being performed on a desktop computer, int-type
10 operations are used. The values "a" and "b" are added together using int-type add. This intermediate value is divided by "c".

Turning now to Fig. 8B, a code sample that illustrates performing an operation that is affected by overflow on operands created by an operation that potentially carries
15 overflow on a resource-constrained computer is presented. Since execution is being performed on a resource-constrained computer, short-type operations are used. The values "a" and "b" are added together using short-type add. The addition creates an intermediate value having overflow from the 16-bit range. This intermediate value is divided by "c". Unlike the addition operator used in Fig. 7B, the division operator is
20 affected by overflow, as shown in Table 2. The 16-bit value is considered to be negative, since the high bit is set. Thus, the desktop computer and resource-constraint computer

examples provide different results that have not been corrected by type conversion expressed in the program as in Figs. 6A-7B.

According to the present invention, arithmetic expressions are optimized using
5 typed instructions that are optimal based upon the types of operands. The optimization process proceeds until a potential overflow problem is encountered. At this point, the input operands of the arithmetic expression are revisited and converted to the next larger typed instructions. This process repeats until the appropriate type of instructions are chosen so that arithmetic expressions render the same result on desktop computers and on
10 resource-constrained devices with optimized instruction sets.

Several relationships are maintained during the conversion process. These relationships relate to instructions and the values that will be produced when executing the instructions on the target machine. The relationship data includes the actual and the
15 desired type for a value. The relationship data also includes the source instruction that will produce the value on the target machine once the instruction is executed on the target machine. Each instruction is also linked to its operand(s) relationship data. Additionally, the relationship data for a result is linked to the instruction(s) that consume the result. Each relationship data is also linked to the instruction (if any) that will cause potential
20 overflow if the instruction is executed on the target machine. This instruction is referred to as a rollback point. Since an erroneous final result may be produced when a value which carries potential overflow is consumed by an operator that is sensitive to overflow,

linking each value that will be produced to the instruction that caused the overflow provides a way to roll back to the instruction that caused the overflow problem when the conversion process cannot proceed further.

5 An intermediate value can be further consumed as an operand in successor instructions. If the intermediate value potentially carries overflow, the rollback instruction is also propagated in the result. This repeats in the course of converting an expression. The rollback action always acts on an intermediate value (or operand) and rolls back to the instruction where a re-conversion is required. A method for determining
10 the rollback instruction and other details of the optimization are discussed below.

Turning now to Fig. 9, a flow diagram that illustrates n-base typed arithmetic expression optimization in accordance with one embodiment of the present invention is presented. At reference numeral 80, an instruction to be converted is received. At
15 reference numeral 82, a determination is made regarding whether any of the input operands carry potential overflow. If at least one operand carries potential overflow, a determination regarding whether the instruction being converted is sensitive to overflow is made at reference numeral 84. The Java™ bytecodes for these are listed in Table 2. Those of ordinary skill in the art will recognize that the list of operators affected by
20 overflow may vary for different high-level languages, and that this invention may be applied to these other languages as well.

At reference numeral 86, if the instruction being converted is sensitive to overflow, the conversion process is rolled back to the instruction that is the source of the problem and that instruction is converted using a type having a wider base. For example, an 8-bit byte type would be widened to a 16-bit word type, and a 16-bit word type would be widened to a 32-bit word type. Widening an operand to a larger type requires subsequent instruction conversions of the operand to use instructions tied to the larger type.

If the instruction being converted is insensitive to overflow, or if none of the input operands carry potential overflow, the instruction is converted to the most optimal type for execution on a resource-constrained device at reference numeral 88. At reference numeral 90, a determination is made regarding whether more instructions remain to be converted. If more instructions remain, conversion of the next instruction begins at reference numeral 80. The conversion process ends at reference numeral 92 when the last instruction has been converted.

Turning now to Fig. 10, a detailed flow diagram that illustrates n-base typed arithmetic expression optimization in accordance with one embodiment of the present invention is presented. At reference numeral 100, an indication that the conversion is not complete is made. At reference numeral 102, an indication that conversion of the first instruction should be performed is made. At reference numeral 104, whether instruction conversion has been completed is determined. If instruction conversion has been

completed, execution terminates at reference numeral 106. If conversion has not been completed, an indication that conversion is complete is made at reference numeral 108. At reference numeral 110, the first instruction is obtained. At reference numeral 112, a determination of whether the instruction should be converted is made.

5

If the instruction should be converted, an indication that the conversion is not complete and an indication that the current instruction should not be converted again are made at reference numerals 114 and 116, respectively. At reference numeral 118, the instruction is converted to another instruction optimized for a target machine having a
10 smaller base type. At reference numeral 120, a determination is made regarding whether a rollback has been triggered by the conversion at reference numeral 118. If a rollback has been triggered, the instruction at the rollback point is obtained at reference numeral 122 and conversion of the instruction at the rollback point is restarted at reference numeral 104. If rollback is not triggered, the result type and the desired type are matched
15 at reference numeral 124. At reference numeral 126, the conversion information is propagated to successor instructions for each control path.

At reference numeral 128, a determination regarding whether more instructions remain is made. If more instructions remain, the next instruction is obtained at reference
20 numeral 130 and execution continues at reference numeral 112. The conversion process ends when the last instruction has been converted.

Turning now to Fig. 11, a flow diagram that illustrates converting an instruction in accordance with one embodiment of the present invention is presented. At reference numeral 140, whether the current instruction is an arithmetic instruction is determined. If the instruction is an arithmetic instruction, it is converted at reference numeral 142.

5 Similarly, stack manipulation, target, type conversion and convert initial value instructions are converted at reference numerals 146, 150, 154 and 158, respectively. The classification of instructions according to whether an instruction is an arithmetic, stack manipulation, type conversion or initial value instruction in Fig. 11 is for illustrative purposes only. Those of ordinary skill in the art will recognize that the
10 invention may be applied to many other instruction types or classifications as well.

Turning now to Fig. 12A, a flow diagram that illustrates a method for converting a target instruction in accordance with one embodiment of the present instruction is presented. In the Java™ Virtual Machine instruction set, target instructions include
15 branch, switch, array access, array creation and variable store/put instructions, as well as any other type-sensitive non-arithmetic instructions in a computer language that is not a stack manipulation, type conversion, initial value instruction or arithmetic expression.

At reference numeral 160, the desired types for instruction operands are
20 determined. At reference numeral 162, a determination is made regarding whether the operands consumed by the target instruction have types that are smaller than the desired types of the target instruction. If the operands have types that are smaller than desired,

the conversion process is rolled back with the smaller typed operand at reference numeral 164. If the operands do not have types that are smaller than desired, a determination regarding whether the operands carry potential overflow is made at reference numeral 166. An operand may carry potential overflow if it was created by one of the operators listed in Table 1, or if it is created by an operator that propagates overflow in an operand. Operators that propagate overflow include, by way of example, the “and”, “or” and exclusive “or” (xor) operators. If none of the operands carries potential overflow, the instruction is optimized at reference numeral 167. If at least one of the operands potentially carries overflow, the conversion process is rolled back with the smaller operand at reference numeral 164.

Turning now to Fig. 12B, a flow diagram that illustrates a method for converting an initial value instruction in accordance with one embodiment of the present invention is presented. Examples of initial value instructions include get/load instructions and other instructions that load a variable. Initial value instructions also include method invocation instructions, which return the method result. Additionally, initial value instructions include load constant instructions. These instructions are called “initial value” instructions because the values produced by the instructions are not the result of an intermediate computation. At reference numeral 168, the type of a variable, returned value or constant is received. At reference numeral 169, the initial value instruction is optimized according to the type of the variable or constant. For example, to load a short typed local variable, the iload instruction is optimized to sload.

Turning now to Fig. 13, a flow diagram that illustrates a method for converting a type conversion instruction in accordance with one embodiment of the present instruction is presented. A type conversion instruction may convert an operand to a larger type or a smaller type. For example, casting a 32-bit int type to a 16-bit short type converts the operand to smaller type. Likewise, casting an 8-bit byte type to a 32-bit int type converts the operand to a larger type. In the latter case, the byte type is called the operand type, and the int type is called the target type.

At reference numeral 170, an instruction is received. The operand type and the target type are determined at reference numerals 172 and 174, respectively. If the operand type is larger than the target type, the operand type is narrowed to the target type at reference numeral 178. If the operand type is smaller than the target type, a determination is made regarding whether the operand potentially carries overflow at reference numeral 180. If the operand potentially carries overflow, the conversion process is rolled back to correct the type at reference numeral 182. If the operand does not carry potential overflow, the operand is widened to the target type at reference numeral 184.

Turning now to Fig. 14, a flow diagram that illustrates a method for converting a stack manipulation instruction in accordance with one embodiment of the present instruction is presented. In the Java™ Virtual Machine instruction set, stack

manipulation instructions include the "dup", "swap" and "pop" instructions. At reference numeral 190, an instruction is received. At reference numeral 192, a determination is made regarding whether the instruction is a dup instruction. If the instruction is a dup instruction, a determination regarding whether a rollback point for the original stack entry exists is made at reference numeral 194. If the original stack entry does not have a rollback point, the rollback point for the duplicated stack entry is set to the rollback point for the original stack entry at reference numeral 196. If the original stack entry has a rollback point, the rollback point for the duplicated stack entry is set to the source instruction of the original stack entry at reference numeral 198. At reference numeral 200, the instruction is converted.

Turning now to Fig. 15, a flow diagram that illustrates a method for converting an arithmetic expression in accordance with one embodiment of the present invention is presented. At reference numeral 210, a determination is made regarding whether the operands carry potential overflow. If the operands do not carry potential overflow, an indication that the operands do not have potential overflow is made at reference numeral 212. If the operands carry potential overflow, a determination regarding whether the instruction is affected by overflow is made at reference numeral 214. If the instruction is not affected by overflow, an indication that the operand has potential overflow is made at reference numeral 216. If the instruction is affected by overflow, the conversion is rolled back at reference numeral 218 to the first operand with overflow. If the conversion is not rolled back, the optimized instruction type is determined at reference numeral 220, the

instruction is optimized at reference numeral 222 and the result type and result overflow are determined at reference numeral 224.

Turning now to Fig. 16, a flow diagram that illustrates a method for determining
5 an optimized instruction type in accordance with one embodiment of the present invention is presented. At reference numeral 230, at least one operand is received. At reference numeral 232, the desired instruction type is set to the largest type associated with the operand(s). At reference numeral 234, a determination is made regarding whether any of the operands have types smaller than the desired instruction type. If at
10 least one operand has a type smaller than the desired type, the smaller operand is rolled back to correct the type at reference numeral 236.

Turning now to Fig. 17, a flow diagram that illustrates a method for determining a result type and result overflow in accordance with one embodiment of the present
15 invention is presented. At reference numeral 240, the result type is set to the instruction type. The Java Card™ result types and overflow indications returned are summarized in tables 3 to 10, below. The tables are organized according to the type of instruction. Each table indicates the result type and the overflow indication based upon the types of one or two operands.

type(A)	and/or	type(B)	Result Type	Overflow
byte	and	byte	short	false
int	or	int	int	false
others			short	true

Table 3 – Addition, Multiplication, Subtraction

type(A)	and/or	type(B)	Result Type	Overflow
byte	and	byte	short	false
byte	and	short	short	false
int	or	int	int	false
others			short	true

5 Table 4 – Division

type(A)	Result Type	Overflow
byte	short	true
short	short	true
int	int	false

Table 5 – Left Shift

type(A)	Result Type	Overflow
byte	byte	false
short	short	false
int	int	false

Table 6 – Right Shift

10

type(A)	Result Type	Overflow
byte	short	false
short	short	true
int	int	false

Table 7 – Negate

type(A)	Result Type	Overflow
byte	short	true
short	short	true
int	int	false

Table 8 – Unsigned Right Shift

type(A)	and/or	type(B)	Result Type	Overflow
int	or	int	int	false
others			short	false

Table 9 - Remainder

5

type(A)	and/or	type(B)	Result Type	Overflow
byte	and	byte	byte	false
int	or	int	int	false
others			short	=overflow(operands)

Table 10 – and, or, xor

10 The use of Java Card™ result types and overflow indications in Fig. 17 are for illustrative purposes only. Those of ordinary skill in the art will recognize that the invention is applicable for other high order languages having other types.

At reference numeral 244, a determination is made regarding whether the result potentially carries overflow caused by using a more optimized instruction. If the result does not carry potential overflow, a determination is made regarding whether any operands propagate overflow at reference numeral 246. If at least one operand propagates overflow or if the result potentially carries overflow, the rollback point of the

15

result is recorded at reference numeral 248 and an indication that the result has potential overflow is made at reference numeral 250.

Turning now to Fig. 18, a flow diagram that illustrates a method for recording a
5 rollback point in accordance with one embodiment of the present invention is presented.
At reference numeral 260, a determination is made regarding whether a first operand has
a rollback point associated with it. If the first operand has a rollback point associated
with it, the rollback point for the current instruction is set to the rollback point of the first
operand at reference numeral 262. If the first operand does not have overflow associated
10 with it, a determination regarding whether a second operand has overflow associated with
it is made at reference numeral 264. If the second operand has a rollback point associated
with it, the rollback point of the instruction is set to the rollback point of the second
operand at reference numeral 266. If neither operand has a rollback point associated with
it, the rollback point of the instruction is set to the source instruction for the first operand
15 at reference numeral 268.

According to a specific embodiment of the present invention, as shown in Fig. 18,
“first operand” refers to the one created first. Setting the rollback point to the source
instruction for the older operand may obviate the need to perform an additional rollback
20 operation for the newer operand, since correcting the types associated with the older
operand may correct types used subsequently by the newer operand.

Turning now to Fig. 19, a flow diagram that illustrates a method for rolling back the conversion process in accordance with one embodiment of the present invention is presented. At reference numeral 270, conversion of the current instruction is preempted. At reference numeral 272, a determination regarding whether the operand has a rollback point. If the operand does not have a rollback point, the rollback instruction is set to the source instruction that created the operand at reference numeral 276. If the operand has a rollback point, the rollback instruction is set to the same rollback point at reference numeral 274. At reference numeral 278, the desired type of the rollback instruction is widened. At reference numeral 280, an indication that the rollback instruction should be converted is made. At reference numeral 282, the conversion process resumes at the rollback instruction. At reference numeral 284, the rollback instruction is converted according to the new desired type.

Turning now to Fig. 20, a flow diagram that illustrates propagating the results of an instruction optimization in accordance with one embodiment of the present invention is presented. At reference numeral 290, a successor instruction is obtained. A successor instruction is an instruction in the same control path as the current instruction, and occurring immediately after the current instruction. Those of ordinary skill in the art will recognize that a single instruction may be part of many control paths.

At reference numeral 292, a determination is made regarding whether the successor instruction has been visited previously in the conversion process. If the

successor instruction has not been visited previously, the conversion information for the successor instruction is set equal to the conversion information for the current instruction at reference numeral 294 and an indication that the successor instruction should be converted is made at reference numeral 296. The conversion information may include
5 the runtime state at the current conversion point. For example, the values created by the current or previous instruction that have not been consumed. The values will be used as operands to successor instructions in the control path. For each value, the type, source instruction and rollback point are recorded. If the successor instruction has been visited previously, the conversion information previously recorded at the successor instruction is
10 merged with the current conversion information at reference numeral 298. At reference numeral 298, a determination regarding whether a value within the merged information has been modified is made at reference numeral 300. If a value has been modified, an indication that the successor instruction should be converted is made at reference numeral 296. This process is repeated for each successor instruction.

15

Turning now to Fig. 21, a flow diagram that illustrates merging conversion information from different control paths in accordance with one embodiment of the present invention is presented. At reference numeral 310, the corresponding inputs for both control paths are compared. At reference numeral 312, a determination is made
20 regarding whether the types for corresponding inputs are different. If the types are different, the input having the smaller type is rolled back at reference numeral 314. This process is repeated for each operand.

Turning now to Fig. 22A, a block diagram illustrating instruction conversion in accordance with one embodiment of the present invention is presented. This demonstrates applying the present invention to an arithmetic expression that can be optimized. Fig. 22A illustrates the conversion process for the Java™ expression

short c = (short) ((short) (a + b) / c)

where the values a, b and c are of type short. The Java™ bytecode sequence for this expression is shown at reference numeral 316.

Instruction conversion begins with the iload_a instruction. Instructions associated with the first, smaller type are used for the load and add instructions. As specified in Table 1, the add instruction creates potential overflow, but the explicit cast to type short at the source level removes the possibility of overflow. The div 330 instruction is affected by overflow as indicated in Table 2. However, no potential overflow is present because of the explicit cast. Therefore, the need to “roll back” to the addition operation to create a larger type does not occur.

To further aid in an understanding of the present invention, the example discussed above will be described in more detail, with reference to Figs. 10 to 21.

The iload instruction is a source instruction. At reference numeral 160, the desired type for the “a” operand is type short. At reference numeral 162, the operand “a” is type

short. At reference numeral 166, the operands do not carry potential overflow because they were loaded directly and thus were not created by an operation that creates overflow. Therefore, a short-flavored instruction is used to convert the iload instruction to an sload_a instruction at reference numeral 167. Similarly, the iload_b instruction is
5 converted to an sload_b instruction.

Next, the iadd instruction is processed. Since iadd is an instruction that may create overflow, a check is made to determine whether its operands carry potential overflow at reference numeral 210. Both operands were loaded directly so they do not carry potential
10 overflow. Hence, the optimized result type is determined at reference numeral 220. At reference numeral 232, the instruction type is set to the maximum operand type. In this example, the maximum operand type is type short because both operand "a" and operand "b" are of type short. Since both operands are the same type as the instruction type, type short is returned as the instruction type at reference numeral 238.

15

Next, the instruction is optimized at reference numeral 222. Since the instruction type is type short, the optimized instruction is "sadd". Next, the result type and overflow indication is determined at reference numeral 224. At reference numeral 240, the result type is set to type short, which is the instruction type. Additionally, an indication that the
20 result has potential overflow is made, according to Table 3. Since the result contains potential overflow, the rollback point for the result of (a+b) is recorded at reference numeral 248. Neither operand has a rollback point, so the rollback point for the result is

set to the source instruction for operand “a” (the first operand) at reference numeral 268.

At reference numeral 250, an indication that the result has potential overflow is made.

Next, the i2s instruction is processed. The i2s instruction is a type conversion
5 instruction. At reference numeral 176, the operand type (short) is compared to the target
type (short). Since both types are the same, the type is narrowed to type short at
reference numeral 178, eliminating potential overflow.

Next, the iload_c instruction is processed. Like values a and b, c is of type short
10 and the iload_c instruction is converted to an sload_c instruction. Next, the idiv
instruction is processed. As specified in Table 2, idiv is an instruction that may be
affected by overflow. The “a+b” operand does not carry potential overflow due to the
explicit source-level cast to short, so the optimized divide instruction type is determined to
be type short at reference numeral 232 and the result type is set to type short at reference
15 numeral 240.

Next, the i2s instruction is processed. At reference numeral 176, the operand type
(short) is compared to the target type (short). Since both types are the same, the type is
narrowed to type short at reference numeral 178, eliminating potential overflow.

20

Finally, the istore_c instruction is processed. Since the desired type is type short
and the operands do not carry overflow, the istore_c instruction is optimized to a sstore_c

instruction at reference numeral 167. The converted bytecodes are shown at reference numeral 318.

Turning now to Fig. 22B, a block diagram illustrating instruction conversion in accordance with one embodiment of the present invention is presented. This demonstrates applying the present invention to an arithmetic expression that cannot be optimized. Nevertheless, the converted code maintains semantic equivalence with the unconverted code. Fig. 22B illustrates the conversion process for the Java™ expression

short c = (short) ((a + b) / c)

where the values a, b and c are of type short. The Java™ bytecode sequence for this expression is shown at reference numeral 320.

Instruction conversion begins with the iload_a instruction, represented at reference numeral 322. Instructions associated with the first, smaller type are used for the load 322, 324 and add 326 instructions. As specified in Table 1, the add instruction 326 creates the potential for overflow, but does not require using the second, larger type. The div 330 instruction, however, is affected by overflow. This is indicated in Table 2. Thus, the instructions creating the overflow problem must be corrected. The problem is corrected by “rolling back” to reference numeral 322 and using the second, large-typed instructions for operand “a”.

At reference numeral 332, instruction conversion proceeds a second time until it is determined that operand "b" must also be converted to a larger type, requiring rolling back a second time. Instruction conversion then proceeds a third time at reference numeral 334 until it is determined that the instructions for operand "c" need to use the larger type. Rollback is performed a third time, the type for operand "c" is corrected and the conversion process completes after continuing the conversion process a fourth time at reference numeral 336.

To further aid in an understanding of the present invention, the example of Fig. 22B discussed above will be described in more detail, with reference to Figs. 10 to 21.

The initial conversion of the iload_a, iload_b and iadd instructions proceeds as described in the previous example. Next, the iload_c instruction is converted to an sload_c instruction at reference numeral 167. Next, the idiv instruction is processed. As specified in Table 2, idiv is an instruction that may be affected by overflow. The "a+b" operand the potential for overflow because it was created by the "+" operator and that operator may create overflow as indicated in Table 1. Since at least one operand the potential for overflow, a rollback to the first operand with overflow is performed at reference numeral 218.

At reference numeral 270, conversion of the current instruction is preempted. At reference numeral 274, overflow is associated with the a+b operand, so the rollback point

is set to the rollback point for the a+b operand. At reference numeral 278, the desired type of rollback instruction is widened from type short to type int. At reference numeral 280, an indication to convert the instruction being rolled back to is made. At reference numeral 282, the conversion process is resumed at the iload_a instruction, which was previously converted to an sload_a instruction. At reference numeral 284, the iload_a instruction is converted.

As a result of the rollback, the iload_a instruction is processed at reference numeral 338. At reference numeral 124, the result type and the desired type are matched. Since the result type is short and the desired type is int, the types do not match. Thus, the S2I instruction is created to promote the short to an int. Processing continues with the iload_b instruction and the iadd instruction. At reference numeral 210, the operands for the iadd instruction do not carry potential overflow, so the optimized result type is determined at reference numeral 220. At reference numeral 234, the operand types are compared. Since the "a" operand is now type int and the "b" operand is still type short, rollback is performed for the "b" operand. At reference numeral 276, the rollback instruction is set to the iload_b instruction 340. At reference numeral 278, the desired type is set to int. At reference numeral 280, an indication to convert the current instruction is made. At reference numerals 282 and 284, the conversion is resumed at the iload_b instruction and the instruction is converted.

At reference numeral 124, the result type and the desired type are matched. Since the result type is short and the desired type is int, the types do not match, Thus, the S2I instruction is created to promote the short to an int.

5 Next, the iadd instruction is processed. After rolling back twice, neither operand has the potential for overflow. Therefore, an indication that the operand does not carry potential overflow is made at reference numeral 210 and the optimized instruction type is determined at reference numeral 220. At reference numeral 232, the instruction type is set to the maximum operand type. Since the a+b operand is type int and the "c" operand
10 is type short, the instruction type is set to int. Since the "c" operand type is different than the instruction type, rollback is performed on the "c" operand at reference numeral 236. At reference numeral 276, the rollback instruction is set to the iload_c instruction. At reference numeral 278, the desired type of rollback instruction is widened from type short to type int. The conversion process resumes at the iload_c instruction 342.

15

At reference numeral 124, the result type and the desired type are matched. Since the result type is short and the desired type is int, the types do not match, Thus, the S2I instruction is created to promote the short to an int.

20

Next, the idiv instruction is processed. At reference numeral 238, the optimized instruction type is set to int, since both operands are of type int. At reference numeral 222, an int-flavored instruction (idiv) is selected. The final instruction sequence is

represented at reference numeral 344 of Fig. 22.

Although the present invention has been described with regard to integral types, those of ordinary skill in the art will recognize that the present invention may be applied
5 to floating-point arithmetic expressions as well. Furthermore, although the present invention has been illustrated with respect to Java Card™ technology, those of ordinary skill in the art will recognize that the invention is applicable to many other platforms. These platforms include, by way of example, K virtual machine (KVM) technology. KVM technology is described in "The K Virtual Machine (KVM) - A White Paper", June
10 8, 1999, Sun Microsystems, Inc.

The present invention may be implemented in software or firmware. It may be implemented in other processors, as well as in programmable gate array devices, Application Specific Integrated Circuits (ASICs), and other hardware.

15

Thus, a novel method for adaptive optimization of arithmetic expressions has been described. Uniform-typed instructions are converted to semantically equivalent typed instructions for a second, smaller (having a smaller number of bits) integral type for execution on a resource-constrained machine, thus providing relatively efficient stack
20 utilization and increased execution speed. While embodiments and applications of this invention have been shown and described, it would be apparent to those skilled in the art having the benefit of this disclosure that many more modifications than mentioned above

are possible without departing from the inventive concepts herein. The invention, therefore, is not to be restricted except in the spirit of the appended claims.